SUPLEMENTO MENSUAL DEDICADO A LA IMAGEN SINTÉTICA

RENDERmanía

Número 8

Informe
Ídolos virtuales en Japón

Cómo...
Crear terrenos con HS-Lab

Taller Virtual
Trees.inc, generación de árboles

Foro del lector
El rincón de los lectores

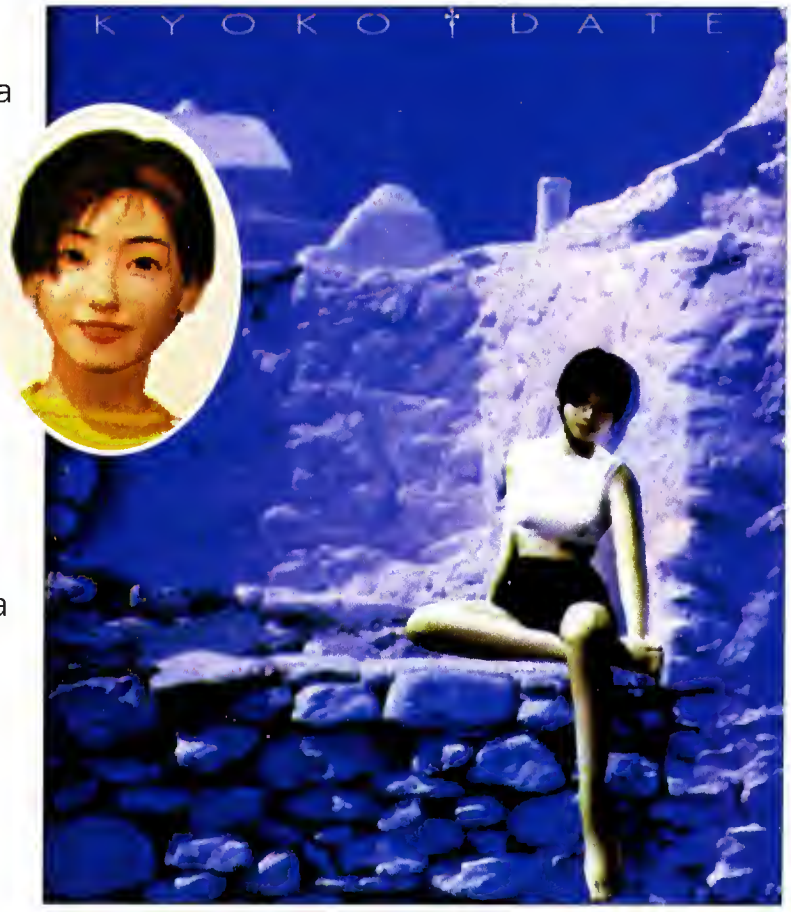
pcmanía

Todo los ficheros de inclusión y los ejemplos podéis encontrarlos en el directorio PCMANIA\RENDER58 del CD-Rom.

José Manuel Muñoz

Ídolos Virtuales

Los "personajes virtuales" son un tema ya añejo en el mundo de la ciencia ficción del que hay múltiples ejemplos: la IA de la novela «La luna es una cruel amante» de Heinlein, el personaje "Max Headroom" de la serie de televisión, la marioneta virtual de Johnny Nnemonic... Los ejemplos son innumerables en el campo de la Sci-Fi. Ahora bien, ¿existen ya personas virtuales en el mundo real?



Todo depende de lo que se acepte como definición de "persona virtual". Es difícil definir este término, pero parece que la idea clásica del mismo es la de una representación fotorealista de una criatura virtual residente en uno o varios ordenadores. En otras palabras, una persona virtual es una Inteligencia Artificial (IA) que usa un interface 3d para representarse a si misma ante el mundo real.

Si damos por válida esta definición, las personas virtuales no existen aún en nuestro mundo real. ¿Existirán algún día? Si aceptamos los criterios del famoso físico-matemático Roger Penro-

El 17 cumpleaños de Kyoko.

Esta chica no es de carne y hueso.

se, el teorema de Godel demuestra que cualquier programa de ordenador está sometido a limitaciones básicas que imposibilitan la aparición de la inteligencia. Otros científicos como John McCarthy –creador del lenguaje Lisp– opinan que, aún aceptando este punto, algún día existirán programas capaces de superar el test de Turing –conocido test ideado por Alan Turing para determinar la posible inteligencia de un programa–.

La opinión del autor de estas líneas es que McCarthy tiene razón: algún día existirán programas inteligentes. De todos modos es curioso constatar que hoy en día ya existen programas no autoconscientes capaces de conseguir resultados que antes se consideraban sólo realizables por personas dotadas de gran inteligencia. Este es el caso de Deep Blue, por su encuentro ajedrecístico con Kasparov, o el de los programas que han desarrollado teoremas matemáticos. Por otro lado, en mi opinión, el test de Turing no debería ser aceptado como válido para medir la inteligencia de un programa por dos razones: la primera porque creo que pueden escribirse programas no inteligentes que pasarían el test, y la segunda porque hay personas que probablemente no lo superarían.

En fin, ya dejando aparte el interesantísimo pero pantanoso campo de la inteligencia artificial, hoy día ya existen actores, presentadores e, incluso, ídolos virtuales. Estas "criaturas" no son, por ahora, más que marionetas virtuales manejadas por sus creadores humanos. Algunas de ellas interactúan en tiempo real con el mundo de los seres humanos mientras que otras, las más "reales", sólo se emplean en la generación de escenas o películas. Veamos algunos ejemplos.

DK-96

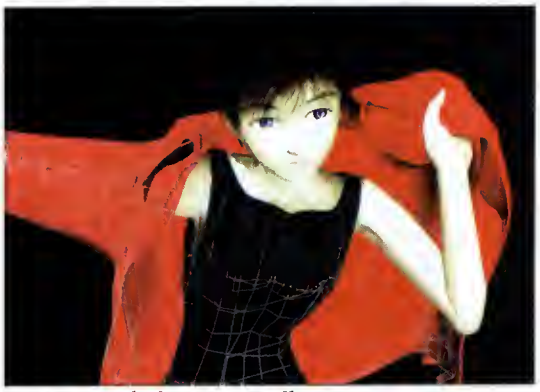
DK-96 o Kyoko Date, como se le suele llamar, es la primera estrella virtual nacida en Japón. Se trata de una chica sintética (pero extremadamente realista) creada por la agencia japonesa de modelos HoriPro, la cual empleó en este proyecto a un grupo de ingenieros y diseñadores que durante cerca de año y medio trabajaron para dar a luz a Kyoko. El web original de HoriPro está en japonés pero, afortunadamente, existe una versión traducida al inglés de sus páginas. La dirección de este web inglés es *www.dhw.co.jp*

Los pocos datos que conocemos sobre Kyoko han sido extraídos de diversos webs dedicados a ella (razón por la que pedimos disculpas de antemano, por si incurrimos en alguna inexactitud). Parece ser que su cuerpo virtual consta de unos 40.000 polígonos y que ha sido elaborado enteramente usando técnicas de modelado 3d en lugar de, por ejemplo, escáneres 3d. En cuanto a la personalidad de Kyoko, sus creadores han sido también muy detallistas: Kyoko tiene 17 años, grupo sanguíneo, familia, amigos, aficiones, etc. Ha grabado varios singles (¡¡!!), se le han hecho diversas entrevistas, y es una buena danzarina. Salvo por el hecho de que no tiene un cuerpo físico, a Kyoko no le falta ninguna de las características propias de cualquier ídolo juvenil. Incluso cuenta con varios clubs de fans entre los seres humanos reales.

Naturalmente, y dado que hoy por hoy no existe ninguna IA digna de este nombre, la personalidad de Kyoko ha sido creada sumando rasgos diversos de varias personas reales. Todos sus movimientos y su manera de bailar se han tomado de una bailarina real, empleando "motion capture", su voz la ha tomado "prestada" de una profesional de la radio y sus singles han sido grabados por una cantante japonesa. Sus expresiones faciales, movimientos oculares, etc, son (¡ay!) únicamente manipulaciones informáticas a las que se apoya de vez en cuando con "motion capture". Todo esto, sin embargo, se olvida rápidamente cuando se ve una foto o una animación de Kyoko, y uno acaba viéndola como un ser real, tal y como sucede con otros personajes de ficción particularmente bien esbozados por sus creadores.

Por si aún queda alguien que no lo sepa, "motion capture" es una tecnología que permite "capturar" los movimientos corporales o faciales de los se-

Rina parece la hermana pequeña de Kyoko.



Rina preparándose para salir.

¿Es Rina otra modelo virtual?

res del mundo real empleando sensores o cámaras. Estos movimientos se convierten en datos numéricos 3d que pueden ser empleados por modelos en 3D. Según parece, en el caso de Kyoko, esto ha sido cuidado hasta el punto de hacer coincidir las expresiones faciales con lo hablado o cantado por esta chica.

El pelo de Kyoko es corto, según se afirma para ahorrar tiempo de proceso y memoria. Las iniciales DK vienen de "digital kids" (chicos digitales) y los "padres" de esta chica esperan que en pocos años la tecnología avance como para permitir que Kyoko participe en Shows televisivos de tiempo real, junto a los hermanos que, sin duda, la seguirán.

En fin, para aquellos que deseéis información adicional, os recomendamos visitar alguna de las direcciones indicadas. Podréis oír cantar a Kyoko, verla bailar, leer alguna de sus entrevistas (su biografía -ay- aún no ha sido traducida del japonés original) o enteraros de algunos de sus gustos. Como curiosidad podemos comentar que su película favorita es Toy Story y su tipo favorito de hombre, Goku de Bola de Dragón. Casi nada.

Rina

Muy poco podemos decir de Rina Kawazuki, ya que todas las páginas que hemos encontrado sobre ella en Internet están en Japonés. Tan sólo constatar que el estilo 3d con el que está creado el cuerpo de esta chica es bastante más manga y menos realista que el de Kyoko. ¿Se tratará de otro ídolo sintético quinceañero creado por alguna compañía japonesa? ¿Será la obra de algún infografista de talento? En fin, mientras esperamos las respuestas podemos, al menos, disfrutar de las fotos. Lo cierto es que se parece a Kyoko como una gota de agua a otra. ¿Será su hermanita menor?

La chica de Tomwoof

A pesar de su inferior nivel de realismo no hemos resistido la tentación de incluir aquí a esta chica a la que Tomwoof -quizá en serio quizá en broma- ha convertido en su candidata al estrellato virtual. Encontramos a esta chica en una de las páginas de Tomwoof junto con una pequeña reseña sobre sus gustos, edad y ambiciones: "ser la mejor estrella virtual posible".

Nota del autor

No hemos encontrado aún ningún ídolo virtual masculino. ¿Será que todos los infografistas del planeta son hombres? Reclamamos la ayuda de las rendermaniacas.

No hemos incluido aquí personajes de videojuegos como Chunli, Ryu (Street Fighter) o Lara Croft (Tomb Raider) porque son conocidos por los lectores y porque su nivel de realismo es menor. Es indiscutible, sin embargo, el hecho de que estos personajes son, en cierto modo, ídolos de popularidad aún mayor que la de la propia Kyoko Date.

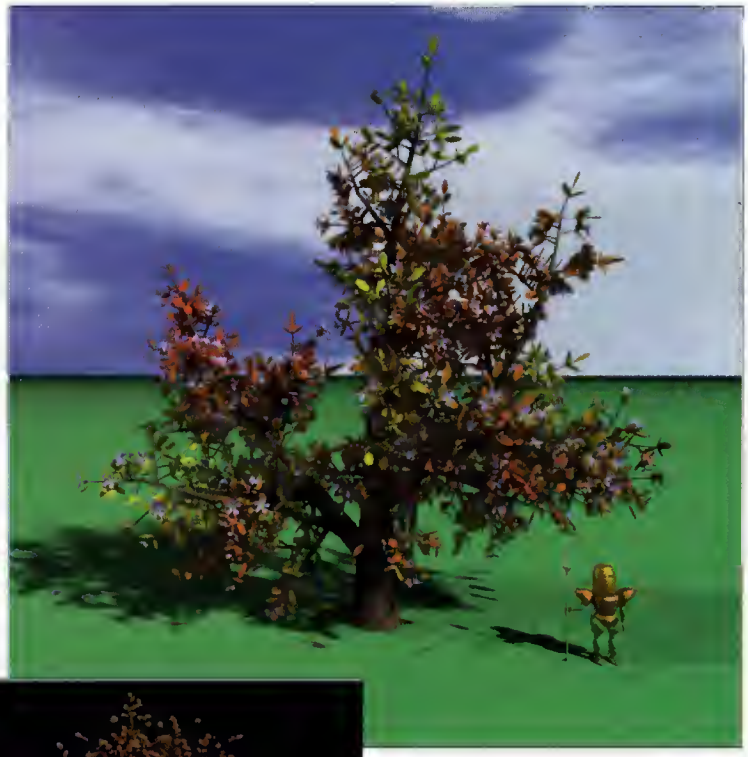
La estrella virtual propuesta por Tomwoof.

Trees. inc:

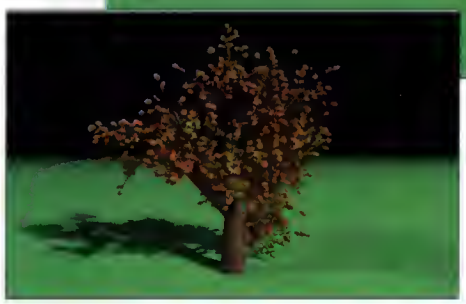
Un Ipas para crear árboles con POV

Las fantásticas sentencias de programación de la versión 3 de POV están propiciando la aparición de toda una serie de "Ipas", como bien podríamos llamar a estas utilidades, para POV. Uno de estos "Ipas" es Trees.inc de Sonya Roberts. Esta utilidad es quizá el mejor camino para crear árboles para nuestras pov-escenas. Con Trees.inc podremos generar árboles de apariencia realista o alienígena, y controlar con facilidad su forma, tamaño, nivel de complejidad, colorido y muchas otras cosas más.

Taller Virtual



En el número 6 de Rendermanía se incluyó una pequeña disertación sobre el funcionamiento y uso de las pov-funciones de usuario o pov-macros (como prefiere llamarlas el autor de estas líneas). Como entonces vimos, una pov-macro es, simplemente, un fichero .inc que contiene un conjunto de sentencias pov. De cara al usuario, las pov-macros funcionan exactamente igual que las subrutinas de programación, siendo empleadas por lo general para crear objetos cuya forma exacta dependerá de los valores que se haya dado a las variables de la pov-macro. Como recordará el lector, el truco consistía en dar unos valores determinados a las variables de la pov-macro, en el fichero pov, y luego usar una sentencia include para "incluir" las sentencias de la pov-macro. Este proceso puede repetirse cuantas veces se desee obteniendo resultados diferentes en función de los valores que se hayan dado a la pov-macro.



Mencionamos todo esto porque la utilidad Trees.inc de Sonya Roberts es una pov-macro. Así, para cada árbol que deseemos crear, habrá que indicar los valores correspondientes para las variables y colocar una sentencia include.

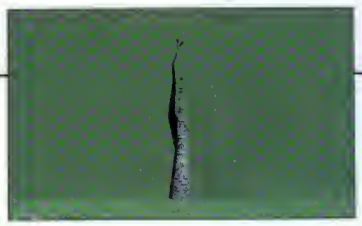
Notas sobre los manuales

A partir de la versión 2.0 de Trees, Sonya decidió escribir un tutorial en formato html (y en inglés) para su utilidad. Algún tiempo después, un usuario realizó una excelente traducción al castellano (también en formato html) que hemos incluido en el CD. Dicha traducción, no obstante, está algo anticuada, ya que se refiere a la versión 2.0 de Trees. La nueva versión de trees, la 3.0, incluye un fichero html adicional y presenta cambios en el apartado "Hojas, flores y frutos" con respecto al antiguo tutorial. Todo lo demás –la explicaciones acerca de cómo se generan los árboles y el funcionamiento de las variables– es prácticamente idéntico en ambos tutoriales. Este artículo se ha escrito gracias a la información suministrada por estos ficheros, la cual ha sido contrastada (claro está) con nuestras propias pruebas.

Los arboles de Sonya

Los árboles generados con el "Ipas" escrito por Sonya Roberts son –como ocurre con los creados por otras utilidades semejantes– modelos formados por cientos o miles de objetos simples. Estos objetos simples son, en su mayor parte, conos y esferas. Los conos se usan para formar el tronco y las ramas, y las esferas se emplean para suavizar las uniones entre los conos. Puede parecer increíble que los árboles que aquí veis se obtengan partiendo de formas tan simples pero el caso es que es así.

Los que os animéis a examinar el fichero Trees.inc observaréis que Sonya ha hecho un experto uso de las llamadas sentencias de programación de POV (if, while, switch, case, rand y otras). Estas sentencias han sido empleadas para escribir los 5 bucles anidados que sirven para generar los 5 niveles de estructura con que cuentan todos los árboles que se construyen con


trees.inc. Estas estructuras son el tronco, las ramas primarias (las que parten del tronco), las secundarias, las terciarias y las terminales, a las que aquí llamaremos ramitas. En cuanto a las posibles hojas, flores y frutos del árbol, sólo pueden partir de las ramitas de éste y el número y distribución de las mismas dependerá, tal y como sucede con el resto de las estructuras del árbol, de los valores que demos a una serie de variables.

La pseudoaleatoriedad es necesaria

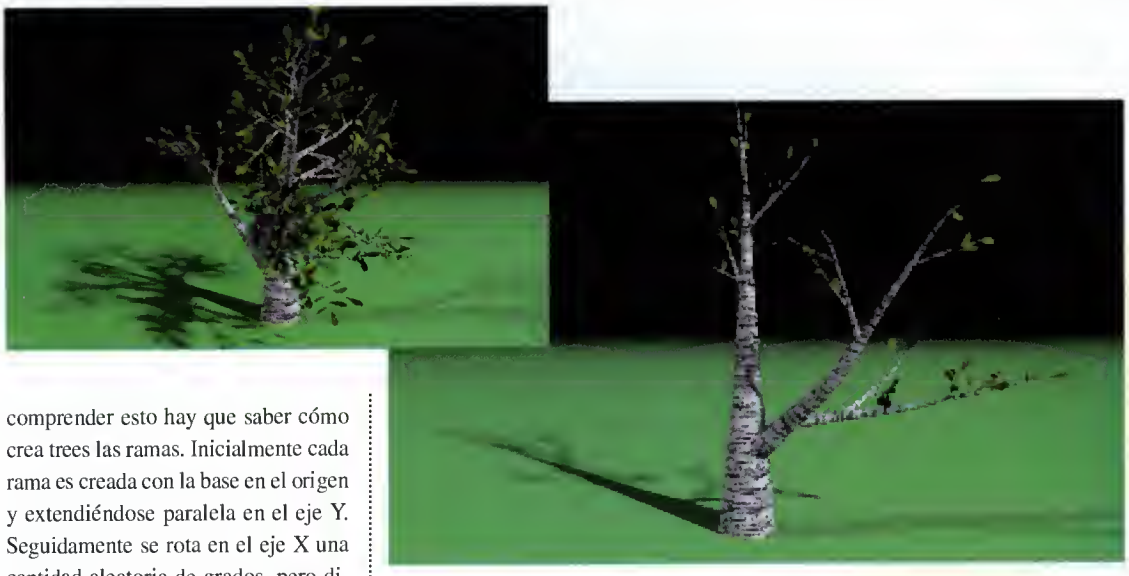
La forma y aspecto finales de cada árbol dependen de una serie de variables que Sonya clasifica en tres categorías: las variables para los generadores de números aleatorios, las variables que controlan la forma y distribución de las estructuras del árbol, y las variables que afectan al tamaño y las proporciones del árbol.

Las variables de la primera categoría sirven, como ya imaginará el lector, para obtener árboles diferentes. Aquí sentimos la necesidad de hacer un inciso en beneficio de aquellos rendermaniacos que desconozcan lo que es un generador de números pseudoaleatorios. Los lenguajes de programación suelen implementar sentencias que retornan números que parecen tomados al azar. Realmente la rutina que constituye el generador emplea una fórmula para generar los números devueltos, con lo cual no puede decirse que estos sean aleatorios. Pero, sin embargo, a un ser humano le resultará imposible predecir a priori los valores que devolverá el generador. Por otro lado el generador necesitará un valor llamado semilla como entrada para ser inicializado. Después de dicha inicialización, y cada vez que sea invocado, el generador nos devolverá el siguiente valor "aleatorio" que corresponda a la semilla suministrada; por esto se llama pseudoaleatorios a estos números. Lo más importante es, como veremos enseguida, que una semilla dada produce siempre la misma secuencia de valores. ¿Para qué puede sernos útil esto? Bien, en infografía existen bastantes aplicaciones en las que es imprescindible el uso de números pseudoaletorios. En el caso de trees, por ejemplo, los valores devueltos por la semilla SD1 se usan para controlar el número de ramas y la orientación de las mismas. Esto significa que bastará con cambiar el valor dado como semilla en la variable SD1 para obtener un árbol diferente. (Y también significa que no podremos predecir si la construcción del árbol nos parecerá satisfactoria. Si no lo es tendremos, simplemente, que probar con otros valores diferentes para la semilla).

En cuanto a la obtención de la misma serie de valores en cada uso de una misma semilla, se trata de algo cuyas ventajas resultan obvias. Por supuesto, no hay ninguna razón para tener dos árboles idénticos en un bosquecillo pero, si estamos creando una animación, esta característica resultará indispensable para no obtener un árbol distinto en cada frame de la misma. Como recordaréis el generador de POV permite emplear varias semillas al mismo tiempo. Dependiendo de cual sea la referenciada (por la sentencia rand), el generador nos devolverá un valor u otro. Gracias a esto Sonya ha podido emplear tres semillas en Trees: además de SD1, que controla las divisiones de ramas en el árbol, trees usa las variables SD2 y SD3. SD2 nos permitirá cambiar la distribución aleatoria de las hojas, flores y frutos en las ramitas sin alterar la distribución general de las ramas ni el número de las mismas (o sea, lo controlado por SD1). SD3 se emplea en algunas cuestiones aleatorias relacionadas con las hojas, flores y frutos.

Variables que afectan a la distribución y a la forma

Aparte de las semillas SD1 y SD2 existe una serie de variables que nos permitirán un control más "predecible" sobre la forma global del árbol. Se trata de las variables que controlan los ángulos de división. Este nombre alude a los ángulos que se formarán en cada eje entre una rama dada y el tronco. Para



comprender esto hay que saber cómo crea trees las ramas. Inicialmente cada rama es creada con la base en el origen y extendiéndose paralela en el eje Y. Seguidamente se rota en el eje X una cantidad aleatoria de grados, pero dicha cantidad se mantiene comprendida entre los valores dados a MinXDeg y MaxXDeg. Podemos dar a estas variables el valor que deseemos pero parece que lo mejor es dar 10 a MinXDeg y poner 30 ó 40 en MaxXDeg.

Después de esta rotación en el eje X, otra rotación aleatoria en el eje Y dará a la rama la orientación adecuada con respecto a la rama padre. De nuevo los valores aleatorios devueltos estarán comprendidos por los números dados a dos variables: MinYDeg y MaxYDeg. Si se desea un reparto uniforme de ramas alrededor del árbol, lo cual es lo normal, los valores respectivos para estas variables deberán ser 0 y 360. Valores diferentes ocasionarán que las ramas sean creadas sólo en una sección del tronco (lo cual puede ser útil si queremos simular un efecto de viento en el árbol). Las variables MinZDeg y MaxZDeg realizan la misma función en el eje Z pero normalmente no son necesarias y podemos dejarlas a 0.

Las tres variables siguientes se emplean para conseguir un interesante efecto: lograr que las ramas se vayan curvando hacia abajo más y más, a medida que se alejan del tronco. Para conseguir esto, los valores dados a las variables IncXDeg, IncYDeg y IncZDeg se suman a los valores mínimos y máximos de las variables explicadas anteriormente. Si por ejemplo hemos dado el valor 5 a IncXDeg, y teníamos los valores 30 y 40 en MinXDeg y MaxXDeg, los valores aleatorios devueltos en el eje X estarán comprendidos entre 30 y 40 grados para las ramas primarias pero pasarán a oscilar entre 35 y 45 grados para las ramas secundarias y las ramas terciarias se inclinarán entre 40 y 50 grados. Podemos dar valores negativos a IncXDeg con lo cual las ramas se curvarán hacia arriba.

Las siguientes variables de este grupo afectan a la frondosidad del árbol. Los valores dados a MinSplits y MaxSplits servirán para especificar el número mínimo y máximo de ramas que pueden partir del tronco o de una rama padre. Hay que tener cuidado sobre todo con el valor dado a MaxSplits, ya que el número de objetos del árbol podría dispararse. Como nuevamente se usa el generador de pseudoaleatoriedad nos será imposible predecir el número total de ramas (a menos que los valores para ambas variables sean idénticos) pero si podemos saber el número máximo de objetos que tendrá el árbol. Sonya adjunta el siguiente ejemplo. Para un valor de 4 en MaxSplits, el número máximo de ramitas será de

4*4*4*4=256

(Sólo hay 4 multiplicaciones porque aunque hay 5 niveles de estructuras-rama, el primer nivel es el tronco).

Otra variable importante es IncSplits. Al usar Trees, el usuario tenderá a dar valores relativamente altos a MinSplits y MaxSplits para obtener árboles frondosos. El resultado final, no obstante, puede ser insatisfactorio si lo que buscamos es conseguir una distribución más "real". Como hace notar Sonya, los árboles reales tienen por lo general más ramas secundarias en cada rama primaria que primarias en el tronco y más ramas terciarias en cada secundaria que secundarias en cada primaria. O sea, que la complejidad estructural de los árboles suele ir creciendo en cada nivel a medida que nos alejamos del tronco. Sonya implementa la variable IncSplits para simular esto. IncSplits es un multiplicador que

afecta a MaxSplits en cada nivel de la estructura. Supóngase que tenemos un valor MaxSplits de 4 y un valor de IncSplits de 1.25 (ejemplo dado por Sonya). Entonces el máximo número posible de ramas primarias será de 4 pero el de ramas secundarias podrá ser de 5 y el de ramas terciarias de 7. Aquí Sonya hace dos recomendaciones: ¡Cuidado con el máximo número posible de ramas! También es recomendable emplear las variables de incremento de grados, ya de lo contrario podría ocurrir que todas las ramas se apelotonarán en un mismo volumen espacial.

Tamaño y proporciones

No es nada recomendable realizar operaciones de escalación una vez construido el árbol ¡tiene demasiados objetos! En lugar de esto daremos el valor adecuado a la variable BaseLen, que se emplea para determinar el tamaño del árbol. BaseLen –cuyo valor por defecto es 1– es el radio de la base del tronco del árbol. Como, por defecto, el árbol tiene siempre las mismas proporciones, cambiando esta variable cambiaremos el tamaño global del árbol. (El tronco del árbol tiene una altura de unos 19 metros).

Otra variable importante es LenghtInc que se usa como multiplicador para modificar las proporciones del árbol. Un valor de 1 da como resultado árboles compactos y un valor cercano a 2 nos da árboles "estirados".

Por último BallJoint controla el nivel de detalle (o número de esferas de suavizado) del árbol. Con un valor de 0 las esferas se desactivan (ahorrando tiempo y memoria) y con un valor de 5 se obtiene el máximo nivel de detalle (para cámaras muy cercanas al árbol)

Hojas flores y frutos

Los cambios más importantes que trae la versión 3.0 de Trees con respecto a su predecesora están en este apartado. Ahora el usuario podrá elegir entre una librería bastante amplia de objetos para adornar sus árboles con hojas, frutos y ramas. E incluso podrá, si es preciso, incluir sus propios objetos para que sean usados con este fin por la utilidad.

Los objetos de la librería son en su mayor parte muy sencillos (la naranja es una esfera simple, el limón es una triple esfera algo alargada, etc.). Esto se debe, por supuesto, a que los árboles se construyen a base de cientos e incluso miles de objetos. Si estos objetos no fueran lo más sencillos posible, entonces el tiempo de render se dispararía (ejem, en realidad ya se dispara). Esto hace que la opción de incluir objetos propios en el árbol no sea demasiado útil. En teoría sería fantástico crear un árbol que tuviese mangababes, coches u otros objetos complejos como frutos. En la práctica, no obstante, el tiempo y las limitaciones de ram lo hacen imposible.

Ahora veamos las variables que indican a Trees qué objetos deben usarse como hojas, frutos y ramas en el árbol, y cuál ha de ser su apariencia. En primer lugar tenemos a LeafShape cuyo valor designará al objeto de la librería que hará las veces de hoja. Su valor puede ir desde 0 a 8. En cuanto a la textura que se aplicará a la hoja seleccionada, será la indicada por el valor que demos a LeafTexture. Después tenemos a FruitShape, que será usado de la misma manera para indicar al objeto de la librería que servirá como fruto. El

valor de esta variable puede estar comprendido entre 0 y 5 y el número de la textura para el fruto se especificará en FruitTexture. Finalmente el valor de FlowerShape se emplea para indicar a Trees el tipo de flor que queremos para el árbol y el valor de FlowerTexture se usa para indicar la textura correspondiente. Hay 6 posibles flores y 6 texturas

posibles entre las que podremos elegir.

En todos los casos el valor por defecto asignado a estas variables es 0 y el valor usado para indicar que va a emplearse un objeto o una textura creada por el usuario es de 1. En este último caso el usuario tendrá que declarar el objeto con el nombre que se indique entre paréntesis en "user-defined" en el fichero tutorial.htm, en el apartado correspondiente. (Nota: no vamos a dar ahora una relación acerca de las formas y texturas de la librería que corresponden a cada valor, ya que Sonya ha incluido unas escenas para ilustrar esto en el fichero ya mencionado).

Seguidamente hay una serie de variables que controlan el número de hojas en cada ramita (con LeafNum) y la orientación de las mismas (con LeafRosette). Dicha orientación forma, por defecto, una espiral a lo largo de la ramita (solamente las ramitas pueden tener hojas, flores y frutos). Poniendo "True" en la variable LeafRandRot lograremos una distribución más irregular y aleatoria (cambiando el valor-semilla de SD2). Por último podemos lograr un bonito efecto de roseta en las hojas poniendo "true" en LeafRosette.

También habremos de especificar qué tipo de objeto deseamos que tenga el árbol al final de cada ramita. Esto se controla con el valor de Tip. Los posibles valores asignables a esta variable –por defecto el valor es 1– son 0 (nada), 1 (hoja), 2 (fruto) y 3 (flor). Con respecto a esto puede suceder que nos parezca un poco irreal que todas las ramitas tengan un objeto en su extremo (aunque sería prácticamente imposible asegurarlo en un árbol medianamente frondoso).

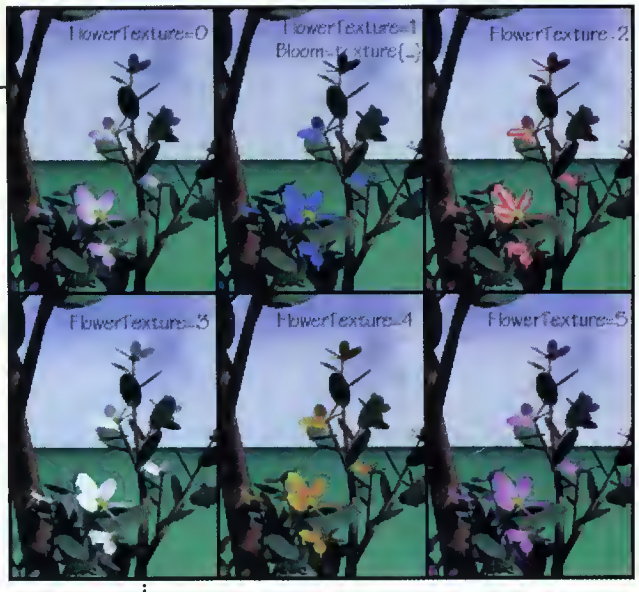
Para indicar a Trees el porcentaje de ramitas que tendrán un objeto en su extremo usaremos la variable TipPercent, dándole un valor oscilante entre 0 y 1. Además podemos emplear la variable

TipOther de manera conjunta con la anterior si deseamos que Trees elija entre dos objetos a colocar al final de la ramita. Para ello indicaremos en TipOther el número del segundo objeto y TipPercent indicará el porcentaje de empleo de uno u otro objeto para las ramitas.

Se nos olvidaba indicar que también hay que indicar el número de textura para la corteza del árbol en la variable BarkTexture. La librería dispone de 6 texturas de corteza (0 a 6) y el valor usado por defecto es el 0.

Texturas propias

Aunque la posibilidad de crear hojas, frutos o flores de formas nuevas no llegue quizá a ser demasiado utilizada por el usuario, si parece muy útil poder definir texturas propias para cualquier objeto del árbol. Esto puede hacerse de dos maneras. La más evidente es asignar la textura definida a, por ejemplo, el tipo de hoja que vaya a utilizarse, pero esto presenta un inconveniente: ¡to-



das las hojas del árbol serán exactamente iguales! No es que esto sea necesariamente malo, sobre todo si el árbol va a estar a cierta distancia, pero podemos lograr un interesante efecto empleando otro método.

Definiremos nuestra propia hoja para el árbol sin asignarle ninguna textura y luego, después del #include "trees.inc", pondremos la textura para las hojas dentro de la declaración del objeto-árbol. Como ya saben los pov-maniacos, de esta manera la textura se aplicará sobre todos aquellos objetos que no tengan declarada su textura propia, o sea, en nuestro caso, sobre las hojas. Naturalmente si la textura del ejemplo es un pigmento simple no habrá ninguna diferencia con respecto al primer método, pero si se está empleando un mapa de color los resultados serán similares a los que pueden verse en el ejemplo del árbol otoñal de la foto con el lancero medieval. Este ejemplo lo encontraréis en lancer.pov y lancer1.pov.

Usando los arboles de Sonya

En una de las pruebas realizadas, el autor de estas líneas decidió crear un árbol cuyas proporciones fuesen congruentes con las del lancero medieval que se creó para estas páginas hace ya tanto tiempo. Como antes se ha dicho, la altura por defecto del árbol es de unas 19 unidades, que aquí aceptaremos como metros. Por tanto para obtener una altura de unos 9 metros bastará con dar un valor de 0.5 a BaseLen. En cuanto al lancero, tenía unas 15 unidades de alto, ya que en las escenas para las que fue originalmente exportado cada metro equivalía a 10 unidades. Por esta razón el modelo fue escalado por 0.10 en todos los ejes.

Por lo que respecta al árbol, se empleó un mapa de color para lograr la diversidad de color deseada entre las hojas –aquí se usó el truco descrito en el apartado anterior–. La escena se renderizó a baja resolución pero luego, al lanzarla al tamaño final, se advirtieron algunos efectos indeseables. En primer lugar, aunque las proporciones generales del árbol eran correctas con respecto al lancero, no ocurría lo mismo con los frutos –limones–, los cuales por su tamaño mas bien parecían melones. Además, el número de limones era excesivo. Por ello, en una segunda prueba, se empleó la variable TipPercent con un valor de .40. Además se copió la definición del limón de Sonya para crear un fruto de usuario, pero escalándolo por 0.50 para que las proporciones fuesen correctas. De todos modos, a pesar de esto, los limones fueron sustituidos por flores en la escena final. ¡Sencillamente ya había demasiados tonos anaranjados en la escena!

Aparte de esto, se uso InsSplits con 1.25 para conseguir el efecto de distribución "natural" de ramas comentado por Sonya y se dio un valor de 8 a IncXDeg para que las ramas se fueran curvando progresivamente. El valor que se ha dejado en SD1 es el que generó el árbol más potable estéticamente hablando.

Algunos consejos

1) Realizad las pruebas iniciales de color, tamaño y proporciones del árbol con valores bajos para MinSplits y MaxSplits. Ahorraréis tiempo.

2) Si queréis recuperar los valores por defecto de Trees para crear un nuevo árbol, incluid el fichero defaults.inc que adjunta Sonya.

3) No escaléis el árbol, usad BaseLen. No rotéis los árboles, dispararéis el tiempo de cálculo.

4) No coloquéis varios árboles demasiado cerca unos de otros. Si sus volúmenes se entrecruzan el tiempo de cálculo se disparará igualmente.

Crear terrenos para

A menudo es conveniente rematar una escena insertando en ella un buen paisaje de fondo. Esto puede ser bastante problemático ya que, aunque existen muchas utilidades que generan "heightfields" para todo tipo de programas de render, no son tantas las que nos permiten tener algún control sobre la apariencia final del paisaje deseado. Este problema también se presenta con nuestro querido POV pero, al menos, podemos elegir entre bastantes herramientas de generación de paisajes para ayudarnos en este asunto. De entre todas ellas quizá sea Hf-lab de John P. Beale la más potente y fácil de usar.

Antes de entrar en las particularidades de Hf-lab nos sentimos obligados a hacer un inciso para explicar algunos conceptos básicos en beneficio de los pov-adeptos más novatos. Los povmaniacos más veteranos pueden saltarse tranquilamente el siguiente apartado.

Los campos de alturas de POV

El lenguaje escénico de Pov dispone de una sentencia llamada height_field que permite al usuario generar paisajes montañosos. Esta sentencia crea un objeto compuesto por una malla de vértices cuyas "alturas" son determinadas por los valores de color de los pixels de un bitmap que el usuario selecciona para tal propósito (y que se suministra como entrada para la sentencia). Naturalmente el fichero con el bitmap no será "pintado" a mano por el usuario (con el auxilio de algún programa de dibujo 2d), sino que será generado por el mismo POV o por alguna utilidad de la que eche mano el pov-adepto.

Otra posibilidad, de la que no vamos a ocuparnos hoy, consiste en importar un objeto con el paisaje ya creado por alguna utilidad. En este caso bastaría con "traducir" el objeto al formato de POV –si ello es necesario– y escalarlo y situarlo debidamente.

El número de vértices que tendrá el objeto que resultará del Height_field dependerá de la resolución del bitmap empleado, ya que cada pixel genera un vértice. Esto no afectará, no obstante, a las dimensiones del objeto, que tendrá 1 unidad de largo en los ejes X y Z.

escenarios de POV

Los vértices que forman la malla montañosa tendrán diversos valores de altura en Y, dependiendo del pixel correspondiente. Estas alturas se generarán o bien a partir de un bitmap de 8 bits –que nos dará 256 posibles alturas– o bien a partir de una imagen de 16 bits –que nos permitirá hasta 65536 alturas diferentes–. Esto quiere decir que los formatos de 16 bits son los más adecuados para nuestros propósitos, ya que nos permitirán obtener una gradación de alturas más suave (o sea, 65536 alturas dentro de un rango de 0 a 1).

El formato de esta sentencia es:

```
height_field {
  Tipo_de_fichero "nombre_fichero"
  [ nivel_del_agua ]
  [ smooth ]
  [ hierarchy ]
}
```

El formato del fichero empleado para el bitmap puede ser gif, tga, pot, png, pgm o ppm. (El formato gif sólo permitirá 256 alturas porque su paleta es de 8 bits). Los parámetros que siguen son opcionales. "Nivel_ del_agua es un valor flotante que deberá estar comprendido entre 0 y 1 y que se refiere precisamente a eso: a la altura a la que supuestamente se halla el nivel del agua en el objeto height_field. Los triángulos cuyos vértices tengan alturas inferiores a la indicada en este parámetro no serán visibles; Pov los ignorará. Por ejemplo, un valor de .5 en este campo le dice a POV que ignore la mitad inferior del objeto.

El siguiente parámetro afecta a la apariencia del height_field. Como ya hemos dicho, éste está compuesto por triángulos y por esta razón la apariencia final del objeto será facetada. Esto, sin embargo, no es algo demasiado deseable en un paisaje montañoso y por ello normalmente incluiremos esta palabra que modifica las normales de las caras del objeto, a fin de conseguir una apariencia más "suavizada".

Resultado de la aplicacición de Pow 3.

En cuanto a "hierarchy", se trata de un parámetro activado por defecto y cuya función parece ser la de acelerar los tests de intersecciones para los cálculos de visibilidad.

Una vez creado, el objeto heigh_field queda dispuesto de tal modo que su esquina inferior izquierda se sitúa en la posición <0,0,0> y su esquina superior derecha queda en <1,1,1>. Por tanto, será conveniente centrarlo con translate antes de escalarlo para que tenga las dimensiones adecuadas para la escena.

Efecto Ring.

El laboratorio de campos de alturas

Sin duda, los lectores más veteranos ya conocerán el truco que puede emplearse con POV para generar heightfields sin recurrir a utilidades externas, un truco que ya hemos mencionado en ocasiones precedentes. Este truco presenta, como tantos otros métodos, el problema de que la forma final del heightfield es poco previsible. Pero, afortunadamente, Hf-lab –la utilidad que adjuntamos en el CD– podrá ayudarnos en esto.

Hf-lab es una utilidad que puede generar, manipular y visualizar heightfields. Se trata, de hecho, de un conjunto de utilidades que han sido agrupadas por Beale. (Entre ellas esta Gforge, creada

Creación de cráteres.

por el propio Beale, y que quizá sea conocida por algunos pov-maniacos). El funcionamiento de Hf-lab se asemeja en gran medida al de un sistema operativo de factura antigua como CP/M o MS-Dos puesto que trabaja mediante comandos escritos.

Estos comandos pueden ser clasificados en varias categorías diferentes. Por un lado tenemos los comandos de generación del heightfield, luego están los operadores y los comandos de manipulación, hay un comando para la visualización del bitmap que corresponde al heightfield en curso y otros para visualizar y manipular al objeto 3d resultante, tenemos también órdenes para manipular el stack y otras de clasificación menos clara.

Instalando Hf-lab

Debemos descomprimir el fichero zip con la orden "Pkunzip -d hf-lab.zip", la cual creará un subdirectorio llamado hf-lab, dentro del cual se descomprimirá la utilidad. Aparte de algunos ficheros de texto y del propio programa veremos un fichero llamado hf-lab.hlp que es la ayuda en línea del programa. Es conveniente instalar dicha ayuda puesto que en ella se explican todos los comandos aceptados por Hf-lab (y que no se explican en los otros ficheros de texto). Para instalar dicha ayuda editaremos nuestro fichero autoexec.bat y colocaremos una línea de la forma:

SET HFLHELP=path_a_hf-lab.hlp

es decir, que si el programa se encuentra en c:\pov301\tools\hf-lab deberemos escribir

SET HFLHELP=c:/pov301/tools/hf-lab/hf-lab.hlp

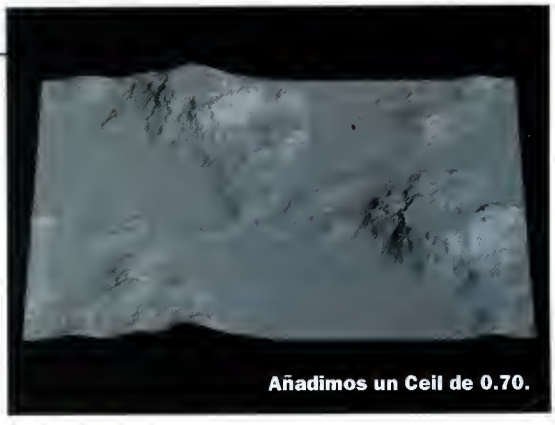
Una vez dentro del programa podremos acceder a esta ayuda en línea tecleando help o "?" para ver una lista de las órdenes permitidas por la utilidad o usando "help comando" para obtener información específica de una orden u operador determinado.

La filosofía de trabajo de Hf-lab

Gran parte de la filosofía de Hf-lab se basa en la idea de la pila o stack. Una idea con la que estarán muy familiarizados los programadores de Ensamblador y los de Forth. Para explicar cómo funciona se suele recurrir a la imagen de la típica pila de platos del fregadero: el primer plato que se colocó en la pila está ahora en el fondo de la misma y el último ha quedado en la cima y será, por consiguiente, el primero en ser fregado. Lo mismo sucede con la pila de Hf-lab la cual tiene, inicialmente, un límite de 4 heightfields. El sistema de trabajo habitual consiste en generar uno o más heightfields con la orden Gforge y realizar diversas manipulaciones en ellos (como añadirles cráteres) para después visualizar los re-

Aumentando el número de cráteres.

Usamos floor con un valor de 0.10.



Añadimos un Ceil de 0.70.

sultados con View. Pasaremos de uno a otro heightfield manipulando la pila y por último, cuando el resultado de nuestro trabajo sea satisfactorio, podremos grabar el bitmap resultante, que luego podrá ser empleado desde POV en alguna sentencia height_field.

El comando gforge

El comando de generación más importante es Gforge y su formato es:

Gforge <n-puntos> <aspereza>

Esta orden crea una malla (o array) cuadrada de n-pixels por lado. La apariencia de la superficie es controlada por el parámetro "aspereza" cuyo valor por defecto es de 2.1; un valor que da una apariencia fragmentada. Si reducimos este valor, la apariencia será algo más suave.

Hay que tener cuidado con el tamaño de los arrays ya que el gasto de memoria va incrementándose a medida que crece la dimensión del array. Podemos realizar pruebas válidas con un tamaño de 256 puntos y luego emplear valores de 512 puntos o incluso más.

Nota: Debido a la forma en que se hacen los cálculos, Hf-lab necesita 8 bytes por pixel durante el proceso de generación. ¡Esto significa que generar una malla de 1000*1000 vértices requerirá 8 megabytes! Por otro lado parece ser que hay un bug perdido en la programación de Hf-lab. Dicho bug produce una fragmentación de memoria Ram similar a la que sucede con el tiempo en los discos duros. Así, a medida que aumenta el número de operaciones realizadas, el programa irá desperdiciando más y más memoria hasta que aparezca el temido mensaje de "out of memory". La única solución que hay para esto es, por ahora, salir del programa y volver a entrar.

Cada vez que creamos un heightfield, la nueva malla pasará a quedar en la cima de la pila, donde podrá ser manipulada y visualizada usando los comandos oportunos –los heightfields anteriores serán desplazados hacia abajo en la pila–. Los comandos "de trabajo" afectan siempre al elemento que se halle en la cima del stack con lo cual, para trabajar sobre otro heightfield, tendremos primero que colocarlo en la cima de la pila, lo que llevaremos a cabo empleando alguna de las sentencias de manipulación de la pila (como "rot", que hace rotar los elementos de la pila, o como "swap" que intercambia el primer y el segundo elemento).

Otro dato a recordar es que cada vez que se imparte una orden gforge, la malla que se producirá será diferente puesto que el programa emplea un generador de números pseudoaleatorios con este fin. Sin embargo podremos generar de nuevo una malla anterior que nos haya gustado si reseteamos el generador suministrándole la misma semilla que dimos para la anterior. Podemos hacer esto escribiendo...

seed valor_de_semilla

En la práctica es conveniente escribir un valor de semilla antes de cada orden gforge, por si se produce un error o queremos recrear parcialmente el desarrollo de un experimento.

Una sesión de trabajo típica

Ahora comencemos la sesión de trabajo. Empezaremos por crear dos heightfields de distintas resoluciones: el primero con 128 pixels y el segundo con 256 (gforge 128 y gforge 256). Ahora escribid "list". Esta última orden listará en pantalla los datos de los heightfields guardados en el stack. La primera línea de datos corresponde a la cima de la pila que, como veréis, está ocupada por el último heightfield creado (el de 256 pixels). Seguidamente teclead "view" para generar una vista tridimensional de este heightfield. El programa dispone de diversas calidades de visualización pero teniendo en cuenta el poco tiempo que precisa el render con la calidad máxima, vale la pena dejar activada esta última. Una vez en el modo "view" podremos impartir diversas órdenes particulares para este modo. Para ello esperaremos a que el render concluya y entonces teclearemos la orden y su parámetro (si lo hay) seguida de return. Así, para regresar al modo normal de texto, tecleareremos "quit" seguido de return. Otras órdenes válidas desde el modo "view" son "S modo_de_sombreado" (con valores de 0 a 3), "G modo_grafico" (que admite 0=320*200, 1=640*480, 2=800*600 y 3=1024*768), y "Tn". Esta última orden parece generar un "tileado" a partir de la malla inicial, o al menos eso es lo


Zedge permite alisar los bordes.

que se aprecia desde el modo 3d (t1=activar el tile, t0=desactivarlo). En el modo de visualización 2d, sin embargo, (orden show) veremos que el bitmap no ha cambiado en nada (¿?).

Ahora vamos a hacer algunos experimentos con la malla pero antes será conveniente preservarla ya que Hf-lab no dispone de undo. Regresad al modo 2d y pulsad "dup". Después ordenad un listado ("list"). Como veréis ha aparecido una nueva malla de idénticas características a la de la cima. Ahora realizaremos algunos experimentos. Primero pulsad "* 2". Esta operación multiplica por 2 la altura de todos los puntos de la malla. Hf-lab nos permite emplear varios operadores como +, -, *, / y Pow, cuyo uso afectará a cada vértice del array. Si el resultado de la operación no nos ha convencido demasiado podremos hacer que el heightfield vuelva a su estado anterior con una división o podremos eliminar el heightfield con la orden "pop", la cual ejerce sobre el stack el efecto contrario al que produce la inclusión de una nueva malla. Con "pop", todos los elementos de la pila a partir del segundo se desplazan hacia arriba con lo que el segundo elemento pasará a ocupar la cima, el tercero pasará a la segunda posición, etc. El primer elemento es, claro está, el situado en la cima de la pila.

Ahora vamos a crear un paisaje con cráteres. Para ello contamos con el comando cráter cuyo formato es...

Crater numero escala_de_profundidad escala_de_radio distancia

El primer parámetro indica el número de cráteres que van a tratarse. Los siguientes se refieren a la profundidad y al radio de los cráteres (pero el autor de estas líneas confiesa que aún no comprende demasiado bien el funcionamiento de estos parámetros. En la ayuda se indica que estos valores son relativos a la normal ¿y?). En cuanto a "distancia" afecta al número total de cráteres que se generarán en el heightfield. El valor por defecto para este parámetro es 10 pero parece más conveniente dejarlo en 5 ó 4 para obtener más cráteres.Usaremos la sentencia, examinaremos el resultado final y –si nos parece adecuado– grabaremos nuestro trabajo escribiendo, por ejemplo, "save crateres.tga". Esto es todo lo que debemos hacer para crear un bitmap utilizable desde POV.


El comando Equalize.

Ahora veamos otros comandos interesantes de Hf-lab.

Con "Floor altura" el programa hará que todas los vértices con alturas inferiores a la especificada por el parámetro tomen la altura de dicho parámetro. El efecto visual será el de crear un plano que simula el nivel del agua en la altura indicada por el parámetro que sigue al comando. Esto puede sernos útil para hacernos una idea del aspecto que presentará el paisaje desde POV, si añadimos al heightfield un plano para simular un mar o un lago a esa altura.

Otro comando del mismo estilo es "Ceil altura", el cual hace lo mismo que el anterior pero igualando los vértices cuyas alturas exceden la indicada por ceil. Esto último puede sernos útil para crear una planicie en una montaña (y colocar algo sobre ella, claro).

Notas del autor

Todas las imágenes del artículo han sido generadas desde POV a partir de las tgas creadas con Hf-lab.

Cada escena ha sido el resultado de un comando aplicado sobre el mismo heightfield inicial. Por cierto, tomad nota de que hay que escalar el heightfield en Y. De lo contrario la imagen que nos dará POV no se parecerá a la previsualización de Hf-lab. Mirad el .pov.

Tan sólo hemos comentado algunas de las órdenes que permite Hf-lab. El programa tiene comandos con los que podremos crear picos, simular la erosión, distorsionar el bitmap de varias maneras, combinar dos heightfields para producir un tercero, etc. Así que ya sabéis: ¡Gforge, gforge y gforge!

El Foro del Lector

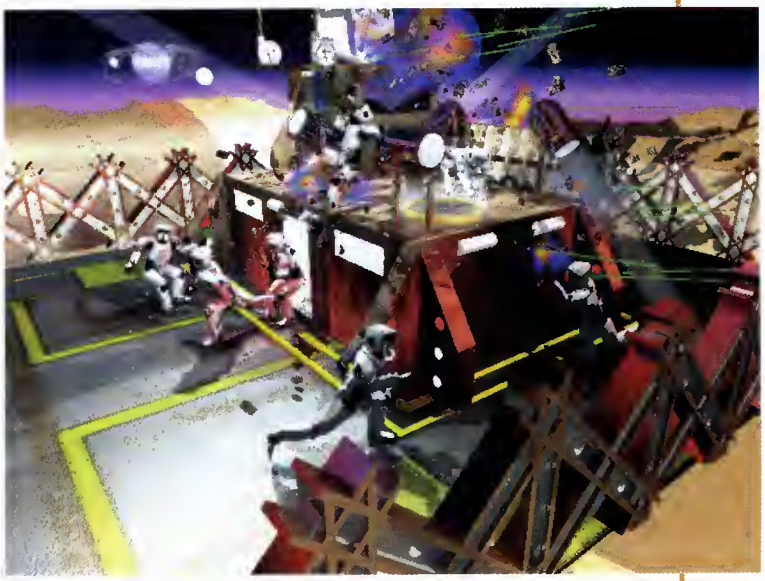
Nota importante. Podéis remitirnos vuestros trabajos o consultas, bien por carta a la dirección que figura en la segunda página de Pcmanía, o vía e-mail a rendermania.pcmania@hobbypress.es

El certamen de Rendermanía

Hoy, aparte del ya habitual carrusel de imágenes enviadas cada mes por los lectores, vamos a presentar una propuesta dirigida a todos los rendermaniacos. Esta propuesta está basada en el éxito que tuvo la idea, inicialmente concebida como una broma, de crear una batalla virtual entre humanos y orcos utilizando modelos enviados por los lectores. Esta propuesta está basada en vuestras propias sugerencias.

Óscar Álvarez Díaz ya apareció en estas páginas en el número 5 con un pov-soldado inspirado en Star Wars. Óscar definió diversas posturas para su soldado y hoy ataca de nuevo con una magnífica escena donde un bunker imperial defendido por sus mejores tropas está siendo masacrado por un... ¿ataque rebelde? Podéis encontrar los detalles de esta obra en la carta de Óscar, en el foro. Estoy de acuerdo contigo en lo que afirmas sobre el manual de POV; muchas veces se echan en falta explicaciones adicionales de algunos comandos y estas lagunas nos obligan a hacer muchas pruebas adicionales. Por lo que respecta a la niebla creo que tus interpretaciones de los comandos para la niebla superficial son correctas, pero no estoy seguro de que sea posible hacer el tipo de humo que pretendías.

En cuanto a lo que dices sobre la generación de paisajes de fondo con POV, estoy parcialmente de acuerdo contigo: no es un asunto fácil (pero si factible). ¿Qué opinas de las dos utilidades "paisajísticas" que incluimos hoy en el CD? Los árboles de Sonya se crean... ¡con sentencias de POV! (apréndetelas ya). Tomo nota de tu sugerencia para dedicar un artículo a la generación de paisajes. Desde mi punto de vista el peor problema es la memoria (los arboles de Sonya o los height-fields ocupan lo suyo) pero incluso para esto existen soluciones que comentaremos en próximos números.



El Foro del Lector

Francisco Javier Díaz Blanco nos envía una imagen creada con 3D Studio, programa con el cual ha tenido dificultades de aprendizaje. ¿Y por qué –pregunta obvia– no te compraste un buen libro sobre este programa? Será difícil que alguien pueda explicarte en pocas palabras el funcionamiento del Keyframer ya que, si echas un vistazo en algún libro dedicado a 3D Studio, comprobarás que la descripción del funcionamiento de este módulo suele requerir capítulos enteros.

En cuanto a mi opinión sobre tu imagen, te diré que me parece excelente teniendo en cuenta el método de aprendizaje que has seguido. Lo único que se te puede criticar, siendo muy puntillosos, es que faltan los caracteres en el teclado y que la aplicación de la textura de madera aparece "corrida" sobre la mesa; hubieras debido dividir la aplicación.

Poco es lo que sabemos sobre esta imagen enviada por el grupo **Epsilon**, salvo que forma parte de un proyecto llamado WildRape. Este grupo solicita correspondencia sobre infografía y programación y nos adjunta su dirección en el universo real que es:

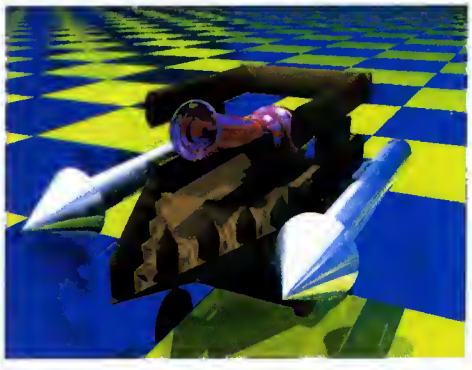
Epsilon
Pº Bera-Bera 57, 1º D 20009 San Sebastián Guipuzcoa
Y su dirección e-mail en el ciberespacio:
epsilon1@arrakis.es

Jorge Albericio nos envía sus primeras pov-escenas: un templo griego y una casa. ¿Y pides crítica y encima constructiva? Pues bien, en la escena de la casa lo primero que llama la atención son las proporciones del conjunto: los bancos parecen desproporcionadamente grandes en relación a la casa. Otro detalle que no convence demasiado es la textura que has empleado para puertas y ventanas. Por otro lado tu obra está muy bien para tratarse de una opera prima, aunque aún es muy pronto para saber si vas a tener futuro en este campo o no. Eso dependerá del tiempo y ganas que pienses invertir y por ello (teniendo también en cuenta lo que ya hay en el mercado) eres tú quien debe saber mejor que nadie si tu futuro va a estar en esto o no. ¡Ánimo y adelante con las siguientes escenas!

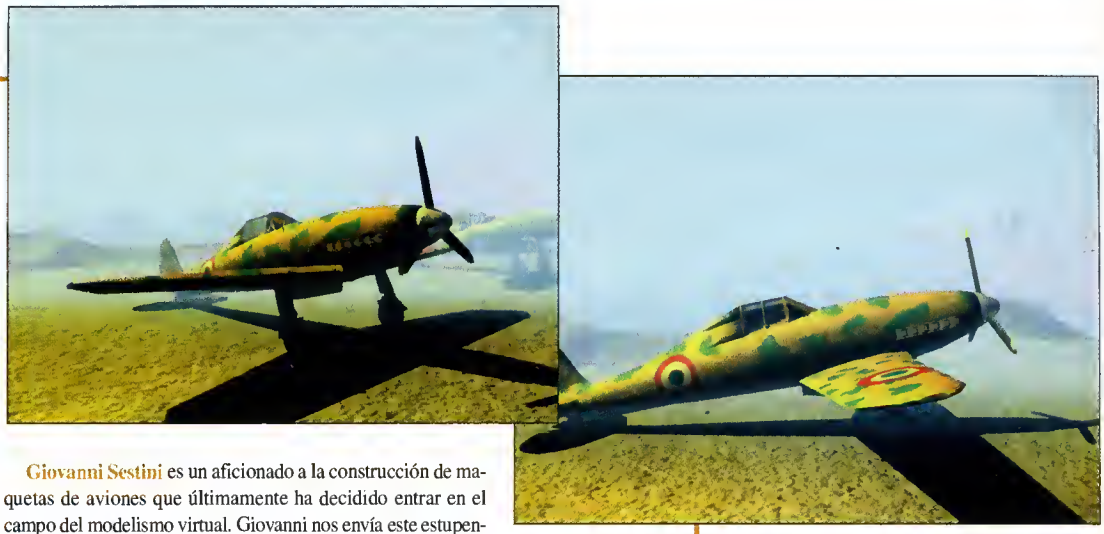
Carlos Monterde Escudero es otro recién llegado al mundo de POV que aúna la pasión por la infografía con el gusto por la programación. Carlos nos envía varias escenas-experimentos donde un objeto mágico –el Cupisferio– suele ser el punto central. Carlos busca colaboradores interesados en realizar videojuegos (leed su carta en el foro) y se confiesa asombrado del nivel de realismo que permite conseguir POV. En cuanto a los problemas que mencionas, para conseguir sombras de bordes difusos utiliza áreas de luz. Esto generará sombras más reales aunque, te aviso, lo pagarás caro en tiempo de render. En cuanto al control de la iluminación, la sentencia scale puede ser empleada para cambiar el tamaño de los objetos a los que se ha asignado una fuente de luz , lo cual no es tu caso. Para controlar la iluminación procura emplear pocas fuentes y limítate a cambiar el valor rgb de las mismas o su posición. De todos modos has logrado una atmósfera muy resultona tanto en Cupsfer.gif como en puerta.gif.

Carlos Pastor Méndez es otro nuevo pov-adepto al que agradezco de corazón que haya decidido apoyar a mis soldaditos en la inminente batalla que han de librar contra los orcos. Tranquilo pov-colega, es cierto que nuestro bando tiene una ligera desventaja en cuanto a variedad de armamento pero no olvides que casi todo el material de los orcos se basa en catapultas mientras que nosotros disponemos de cañones (¡Je!). Además es posible que antes de la batalla se incluya alguna cosilla extra para vapulear a esos malolientes hijos de Gorko y Morko.

En cuanto a tu carro, la idea me parece excelente aunque, no te ofendas, su acabado actual recuerda bastante a la manufactura snotling: Los misiles son demasiado grandes (y además no están permitidos por la convención Orco-humana) y la forma del cañón recuerda a la de un trabuco. Por lo demás la forma general es bastante simpática. ¡Bienvenido a las filas humanas!



El Foro del Lector



Giovanni Sestini es un aficionado a la construcción de maquetas de aviones que últimamente ha decidido entrar en el campo del modelismo virtual. Giovanni nos envía este estupendo "Macchi MC202" creado principalmente con Caligari, programa para el que pide un mayor espacio en Rendermania (tomo nota). Giovanni también pregunta por algún foro de news dedicado al tema del modelismo virtual o por alguna dirección de internet dedicada a este asunto. ¿Puede alguien dar algún dato sobre esto? En cuanto al método de construcción, Giovanni utilizó un escáner para obtener los perfiles del avión. Luego, estos perfiles fueron depurados con Coreldraw antes de grabar las correspondientes plantillas Dxf que se emplearían después. Las piezas fueron montadas con Caligari y 3D Studio y las texturas fueron creadas con Corel Photo Paint, utilizando como plantillas imágenes generadas con las vistas del avión. A pesar de tus notas (excesivamente rigurosas) sobre las carencias del avión, éste nos ha parecido fabuloso.

Pablo, un programador de C++ nos remite las siguientes preguntas:

1)¿Hay alguna tool o librería para manejar imágenes de POV desde C++?

2) ¿Se pueden programar juegos en 3d en el lenguaje de POV?

3) ¿Hay algún libro que explique las prestaciones del lenguaje de POV?

Supongo que no te estás refiriendo a funciones para manejar los archivos de imagen generados por POV desde C++, sino a funciones para manipular el universo 3d de POV desde C++. Existen librerías para leer y manipular archivos gráficos de diversos formatos desde C++, pero no resulta posible controlar POV desde C++. El lenguaje de POV es un lenguaje "escénico", esto es, se trata de un lenguaje diseñado para describir modelos y universos 3d. Es cierto que últimamente –desde la versión 3.0– ha empezado a desdibujarse la frontera entre el lenguaje de POV y cualquier verdadero lenguaje de programación. Así, POV ya admite bucles, ifs, while, rand, etc, pero sigue presentando muchas limitaciones con respecto a los lenguajes convencionales. Además POV emplea la técnica del trazado de rayos para representar las imágenes descritas por los ficheros escénicos con lo cual no resulta posible programar un juego con él ya que puedes necesitar horas o días para generar cada frame. En cuanto a la documentación, puedes echar mano al "Raytracing II" de Anaya (que no trata las sentencias de "programación" de POV puesto que sólo cubre la versión 2.0) o bien puedes leer los números 0 y 1 de Rendermania. No hay por ahora otras alternativas, salvo leerse el manual.

Muchos lectores nos han solicitado documentación sobre el formato asc de 3D Studio. Como de momento no pensamos tratar este tema os remitimos al Web de Viewpoint donde encontraréis documentación sobre éste y muchos otros formatos 3d. No olvidéis que este tipo de documentación suele estar escrita por usuarios que se han estrujado los sesos para obtener la información y no por los diseñadores de los programas. Por esta razón la información suele no ser fiable o completa al 100%.

Anuncio y propuesta de Rendermanía

De ahora en adelante, y siempre que la respuesta que deis a esta propuesta lo permita, intentaremos crear una portada cada cuatro meses empleando modelos enviados por los rendermaniacos. Esta portada estará dedicada a algún tema concreto que se hará público al comenzar cada periodo y, por supuesto, los modelos que enviéis deberán ceñirse al tema tratado. Las escenas de prueba con estos modelos serán publicadas en el foro, donde también se seguirán publicando los trabajos de los lectores que no participen en los certámenes.

En cada convocatoria y dependiendo del tema propuesto, podrán proponerse reglas especiales: el uso de tal o cual utilidad, algunas limitaciones, etc. Habrá, sin duda, algunas dificultades ya que cada uno deseará emplear su programa favorito y habrá que realizar conversiones para traducir cada modelo al render empleado para generar la portada. ¡De vosotros depende el éxito de esta idea! La batalla entre orcos y humanos tendrá lugar en el número siguiente. Entonces anunciaremos las reglas para la primera portada-común y daremos un tema. Se admiten sugerencias: batallas de mechs, escenarios espaciales, ciudades futuristas... ¿Qué preferís?